Orchestrating Masses of Sensors: A Design-Driven Development Approach

Milan Kabáč
Inria Bordeaux, France
milan.kabac@inria.fr

Charles Consel
Inria Bordeaux & University of Bordeaux, France
charles.consel@inria.fr

Abstract

This paper proposes a design-driven development approach that is dedicated to the domain of orchestration of masses of sensors. The developer declares what an application does using a domain-specific language (DSL). Our compiler processes domain-specific declarations to generate a customized programming framework that guides and supports the programming phase.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—domain-specific architectures, languages, patterns

Keywords Generative programming, domain-specific languages, programming frameworks, sensors, pervasive computing

1. Introduction

Masses of sensors are being deployed at the scale of metropolitan areas, country-wide transportation infrastructures, and campuses of buildings. Examples include smart parking lots across entire cities that can direct drivers to available parking spaces, and pollution sensors that are distributed over a city to measure pollution levels and warn frail persons. In this context, the key challenge is to harness the potential benefits of such infrastructures by providing users with innovative and useful services. To achieve this goal, developing software is a crucial activity that enables exploring the scope of potential services, anticipating and responding to users' needs. Developing applications that orchestrate masses of objects raises major challenges because of the scale at which this orchestration takes place. Let us introduce the main challenges by reviewing the typical conceptual phases of an application in this domain, namely service discovery, data gathering and actuating.

Service discovery. In contrast with standard service discovery that addresses individual objects, masses of sensors demand a high-level approach to designating subsets of interest. Specifically, selecting objects of interest among a myriad of objects should be tamed by application-specific abstractions that provide meaningful constructs for grouping sensors. For example, an application may need to manipulate parking spaces at the level of lots or districts. The developer should be able to directly express these application-specific concepts. Beyond expressiveness, when considering masses of sensors, the scalability of a service discovery mechanism is critical to making an orchestrating application usable. In this context, exploiting information about the application behavior is essential to reduce the cost of such activites as service discovery and data gathering, as shown by various works [6, 7]. Furthermore, it has been shown that a mismatch between the application behavior and the network routing algorithms can result in poor performance [6].

Data gathering. Delivery models used to gather data must accommodate masses of sources. For example, applications may require data to be pushed from any number of CO_2 sensors located in underground parking lots, when a given air pollution level is reached. Delivery models have a direct impact on the structure and the logic of an application. Besides, making explicit the delivery models used by an application can be valuable information to ensure an optimal routing structure of the underlying sensor network [7].

Actuating. Processing data may result in taking actions by actuating devices. For example, computing the number of available spaces in parking lots allows to periodically update this number on the entrance screen of each lot.

1.1 Our Approach

To address the challenges examined earlier, we propose a software development approach that covers all the phases of an orchestrating application. To do so, we introduce a domain-specific language dedicated to designing orchestrating applications. Design declarations are then processed to support and guide the programmer using generative programming. This strategy allows to abstract over the characteristics of the sensor network.

Domain-specific design language. To cope with the many dimensions of the orchestration of masses of sensors, we introduce a design language that is dedicated to this domain, allowing the developer to declare what an application does, prior to programming it. This design language, named *DiaSwarm*, consists of constructs dedicated to manipulating objects at a large scale. For example, it provides high-level constructs to declare delivery models of sensors at design time.

Design-specific programming frameworks. We have developed a compiler for DiaSwarm that produces programming support customized with respect to a given DiaSwarm design. This programming support takes the form of a programming framework [4]. For example, the DiaSwarm compiler generates code that gathers data from sensors with declared delivery models, allowing the developer to concentrate on what to do once the data is gathered.

1.2 Our Contributions

DiaSwarm. We introduce a design language dedicated to the domain of orchestrating masses of objects. This language provides high-level, declarative constructs that allow a developer to deal with masses of objects at design time, prior to programming applications.

Compiler. We have developed a compiler that generates programming frameworks dedicated to DiaSwarm designs. These programming frameworks provide high-level support to the developer, while ensuring that programming is driven by the design.

2. DiaSwarm

Our presentation of DiaSwarm focuses on the aspects pertaining to orchestrating objects in the large. The other aspects are inspired by a design language, named DiaSpec, dedicated to traditional pervasive computing environments (*e.g.,* homes, offices) and introduced by Cassou *et al.* [3].

2.1 Working Example

Throughout this paper, we illustrate our approach using a parking management system, whose purpose is to monitor the occupancy of parking lots and regulate the flow of traffic to direct cars to available parking spaces. The working example is inspired by existing smart city projects [1]. In our scenario, we envision an infrastructure capable of monitoring the availability of parking spaces. Sensors measure magnetic field variations to determine whether a parking space is occupied by a car. They are encapsulated inside a waterproof casing, buried underground, and emit their status at regular intervals. The application gathers values from these sensors and provides drivers with the number of available parking spaces for a given parking lot by displaying this information on a screen at the entrance of the lot. In addition, the application suggests parking lots to drivers entering the city in an attempt to optimize the flow of traffic. In this case, suggestions are being broadcast to drivers via panels located at the entrances to the city. Furthermore, the application processes sensor data acquired over a period of 24 hours to determine the average occupancy of a parking lot. Parking managers are kept informed about the occupancy level of a parking lot via messages (*e.g.,* email, text messages).

2.2 Device Declarations

An infrastructure relies on numerous objects that allow applications to determine the current state of the environment and to execute actions accordingly. We refer to these elementary building blocks as devices, whether they are hardware (*e.g.,* sensors) or software (*e.g.,* web services). A device declares its ability to sense the state of the environment as a source. Also, a device may have an action facet that comprises a set of operations that can alter the current state of the environment. Device properties (*e.g.,* id, location, *etc.*) allow device instances to be distinguished from each other; they are called *attributes* and are defined at deployment time. Finally, device declarations offer inheritance, promoting the reusability of sources, actions and attributes. Figure 1 shows device declarations for the parking management system. Line 1 declares the `PresenceSensor` device, which consists of an attribute (line 2), defining the location of the parking space it is associated with. This device only declares one source of information (line 3): a boolean value indicating whether a car is present at the space associated with a sensor. Two actuators are defined in lines 5 and 7. Each class of actuator defines a location attribute specific to its purpose (*i.e.,* parking lot and city entrance). Both share a method to display information (`update` – line 4). Likewise, the `Messenger` actuator (line 9) declares a method (`sendMessage` – line 9) to provide parking managers with information about parking lots.

2.3 Application Design

In our target domain, applications can be seen as interacting with an external environment to measure its state via sensors and modify it via actuators. For such a domain, the application logic is naturally expressed with a Sense/Compute/Control (SCC) paradigm, depicted

```
1  device PresenceSensor
2    { attribute parkingLot as ParkingLotEnum;
3      source presence as Boolean; }
4  device DisplayPanel  { update(status as String); }
5  device ParkingEntrancePanel extends DisplayPanel
6    { attribute location as ParkingLotEnum; }
7  device CityEntrancePanel extends DisplayPanel
8    { attribute location as CityEntranceEnum; }
9  device Messenger { sendMessage(message as String); }
10 enumeration ParkingLotEnum { A22, B16, D6,...}
11 enumeration CityEntranceEnum { NORTH_EAST_14Y, SOUTH_EAST_1A,...}
```

Figure 1. Device declarations.

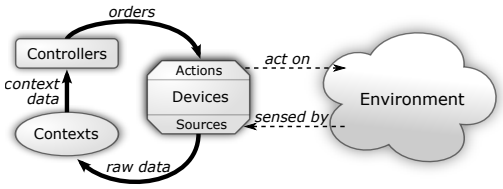

Figure 2. The Sense/Compute/Control paradigm.

in Figure 2. The SCC paradigm, promoted by Taylor *et al.* [8], is general enough for orchestrating objects both in the small and in the large. Consequently, this aspect of DiaSwarm reuses the way DiaSpec declares the design of an application [2]. Specifically, a design consists of (1) declarations of components and devices and (2) descriptions how they interact with each other, forming an acyclic, directed graph from sensors to actuators. DiaSpec introduces two types of components: contexts and controllers. Context components interact with device sources; they receive raw data from the devices, via their sources. They refine (*e.g.,* filter, aggregate) this data into application values, possibly interacting with other context components. When the environment needs to be acted on, a context component declares an interaction with controller components. These components are invoked with refined values and determine what and how actuators are to be invoked.

Figure 3 presents a graphical view of the parking management application in the SCC paradigm. The application declares the `PresenceSensor` device, which produces presence values via its `presence` source to the `ParkingAvailability`, `ParkingUsagePattern` and `AverageOccupancy` contexts. The `ParkingAvailability` context computes the number of available parking spaces in parking lots. This information is passed to the `ParkingEntrancePanel` controller; it is in charge of refreshing the number of available spaces. To do so, this controller component invokes the `update` method of the display panel at the entrance of parking lots.

The `ParkingSuggestion` context provides a list of suggestions of parking lots, based on the information computed by the `ParkingAvailability` component and the usage statistics of parking lots, accumulated by the `ParkingUsagePattern` component. The list of suggestions is passed to the `CityEntrancePanel` controller that administers display panels located at the entrances of the city. The `AverageOccupancy` context calculates the average occupancy of individual parking lots and passes this information to the `Messenger` controller, which notifies parking managers by sending a message via the `Messenger` device.

As can be seen in Figure 3, at a high level, the design of our parking management application does not depend on whether masses of sensors are involved. However, as we examine this application further by presenting the declarations of its constituent components, the need to account for masses of sensors becomes evident, calling for specific constructs. This situation first arises when considering how a context can gather data from a large number of sensors.

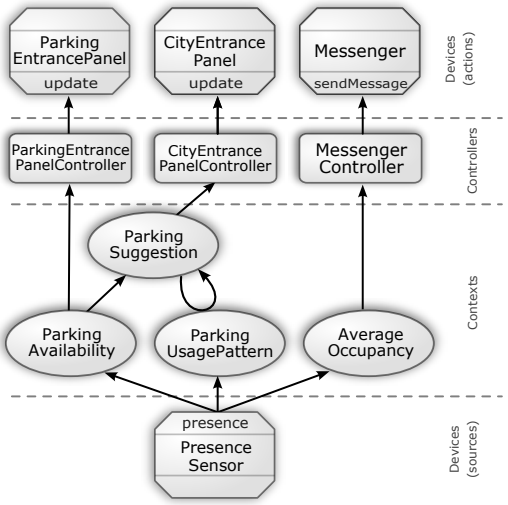

Figure 3. Graphical view of the parking management application.

Data gathering. Because of the nature of our domain, context components mostly gather information from objects at a large scale. To cope with this dimension, our declarative approach provides three data delivery models, inspired by the domain of wireless sensor networks [9], namely *periodic*, *event-driven* and *query-driven*.

Let us illustrate these three data delivery models with our working example and its DiaSwarm declarations given in Figure 4. A context is declared with the keyword `context`, as illustrated in line 1 with the declaration of the `ParkingAvailability` context, whose output type is a sequence of values of type `Availability`. Next, line 2 defines how this context component interacts with its input sensor, namely, `PresenceSensor`. Specifically, the data delivery model for this context is defined as `periodic`. Indeed, recall that presence sensors are assumed to send their status periodically. Thus, our declaration specifies that the `ParkingAvailability` context must be activated following a periodic model, every 10 minutes (*i.e.,* `<10 min>` with presence values). However, the application and the myriad of presence sensors are managed independently. This means that values from the presence sensors are gathered at the sensors' pace, and values are pushed to the application at the application's pace, specified by the context declaration. If the context is faster than the sensors, it will be activated with the same values. If it is too slow, it will miss values. This latter case is illustrated by the `ParkingUsagePattern` that collects parking space occupancy every hour (line 7), as opposed to every 10 minutes, because usage patterns can be determined from coarser-grained information. Finally, the `AverageOccupancy` context determines the average occupancy of a parking lot by processing sensor data acquired over 24 hours (line 15).

DiaSwarm also offers two other delivery models: *event-driven* and *query-driven*. They are denoted by `when provided` and `when required` activation conditions, respectively. The `ParkingSuggestion` context requires data from the `ParkingAvailability` and `ParkingUsagePattern` contexts to produce a list of suggestions of parking lots. This list is computed when the `ParkingAvailability` context outputs a result (see line 19). In fact, all components declared as interacting with the `ParkingAvailability` context will be invoked whenever it produces a value. How the `ParkingAvailability` context produces values is declared in line 4: `always publish`. This construct specifies that the context must publish an output to subscribed components whenever it is activated (*i.e.,* every 10 minutes). The second input to the `ParkingSuggestion` context

```
1  context ParkingAvailability as Availability[] {
2    when periodic presence from PresenceSensor <10 min>
3    grouped by parkingLot
4    always publish;
5  }
6  context ParkingUsagePattern as UsagePattern[] {
7    when periodic presence from PresenceSensor <1 hr>
8    grouped by parkingLot
9    no publish;
10
11   when required;
12 }
13 context AverageOccupancy as ParkingOccupancy[] {
14   when periodic presence from PresenceSensor <10 min>
15   grouped by parkingLot every <24 hr>
16   always publish;
17 }
18 context ParkingSuggestion as ParkingLotEnum[] {
19   when provided ParkingAvailability
20   get ParkingUsagePattern
21   always publish;
22 }
23 controller ParkingEntrancePanelController {
24   when provided ParkingAvailability
25   do udpate on ParkingEntrancePanel;
26 }
27 controller CityEntrancePanelController {
28   when provided ParkingSuggestion
29   do update on CityEntrancePanel;
30 }
31 controller MessengerController {
32   when provided AverageOccupancy
33   do sendMessage on Messenger;
34 }
35 structure Availability {
36   parkingLot as ParkingLotEnum; count as Integer;}
37 structure UsagePattern {
38   parkingLot as ParkingLotEnum; level as UsagePatternEnum;}
39 structure ParkingOccupancy {
40   parkingLot as ParkingLotEnum; occupancy as Float;}
41 enumeration UsagePatternEnum { HIGH, MODERATE, LOW }
```

Figure 4. The design of the parking management application.

is the `ParkingUsagePattern` context. The interaction with this context is query-driven, as denoted by the declaration `get` used in line 20. In fact, the `ParkingUsagePattern` context never publishes values (see line 9). It is assumed that its clients request values from it. This activation condition is expressed by the `when required` declaration.

Service discovery at design time. Discovering in the large requires high-level constructs that are application-tailored. To achieve this goal, we propose discovery constructs that leverage application-specific design concepts. Specifically, DiaSwarm offers the `grouped by` construct parameterized by an attribute. For example, in line 3, the `ParkingAvailability` context requires grouping presence statuses in parking spaces by parking lot, enabling availability to be computed for each lot.

Note that in DiaSwarm, service discovery is part of the design phase, contrasting with existing service discovery that are part of the programming phase [10]. This is a key feature to achieve scalability, as discussed later. Furthermore, because our service discovery approach is global (*i.e.,* not specific to individual sensors), it abstracts over sensor failures; this aspect is delegated to an underlying middleware layer.

Actuating. The declaration of a controller component begins with the `controller` keyword followed by its name. A controller is activated exclusively by the `when provided` condition. For example, the `ParkingEntrancePanel` controller (line 23) is activated by the `ParkingAvailability` context (line 24), causing the `update` action to be triggered on the `ParkingEntrancePanel` device (line 25).

3. Programming Framework

DiaSwarm designs are processed by a compiler that generates customized programming frameworks, currently written in Java. These frameworks provide domain-specific functionalities, including service discovery, data gathering and component interaction. To connect the design phase to the programming phase, the DiaSwarm compiler generates an abstract class for each component declaration. The application logic is implemented by subclassing each abstract class, which in turn requires the abstract methods to be implemented by filling these placeholders with code. The support for devices is examined in our previous work [3].

3.1 Context Components

We start by examining the implementation of the `ParkingAvailability` context component, shown in Figure 5. The developer extends the generated `AbstractParkingAvailability` class with the `ParkingAvailability` class. This subclassing requires the developer to implement a callback method (*i.e.,* `onPeriodicPresence`) that receives the data gathered from the presence sensors, in conformance with the DiaSwarm declaration. Because of the `grouped by` directive, the callback method receives a list of parking spaces indexed by the `parkingLot` attribute. This directive is compiled into a map, which holds entries of the `<ParkingLotEnum, List<Boolean>>` key-value type (line 5), allowing the developer to focus on the data treatment. This treatment is performed by a `for` loop (line 8) over this map. Each iteration processes the parking spaces of a given parking lot. Each entry holds a list of values, indicating the availability of individual parking spaces in a parking lot. In our example, we simply count the number of available parking spaces for each parking lot (line 11). Our example implementation of `ParkingAvailability` returns a list of counts of available parking spaces, indexed by parking lot identifiers, which matches the type of the component declaration. As can be noted, our generative approach allows the developer to abstract over how sensed data are gathered. In particular, the `onPeriodicPresence` method can be implemented without knowing the frequency at which sensors emit measurements, and how many sensors are involved.

3.2 Controller Components

The role of a controller component is to trigger actions on devices to alter the current state of the environment. Similar to a context, a controller is implemented by subclassing the generated abstract class, as illustrated in Figure 6. The generated abstract class `AbstractParkingEntrancePanel` ensures that the `ParkingEntrancePanel` controller receives data from the `ParkingAvailability` context in conformance with design declarations. As a result, the `ParkingEntrancePanel` controller will be notified via the `onParkingAvailability` callback method (line 4) whenever the `ParkingAvailability` context publishes the availability of parking lots. The `onParkingAvailability` method is implemented by overriding the generated abstract method. The arguments passed to the callback method comprise context data (`parkingAvailability`) and the `discover` object. This object is set by the programming framework according to which actuators (and operations) were declared as interacting with this controller component. As shown in line 9, the `discover` object is used to access the display panel of each parking lot. To do so, the `discover` object returns a collection of proxies, wrapped inside a composite object, following the composite design pattern [5]. A proxy provides a means to invoke a remote device, without the need to manage distributed systems details.

4. Conclusion and Future Work

We have presented DiaSwarm, a design language dedicated to the domain of applications orchestrating masses of sensors. We have introduced domain-specific declarations that express the key aspects of such applications: service discovery, data gathering, and actuating. We have illustrated our approach with a working example that exercised the salient features of our language. In the future, we plan to investigate how design declarations can be used to expose parallelism to implement efficient strategies for processing large amounts of data collected from sensors.

```
1  public class ParkingAvailability
2                  extends AbstractParkingAvailability {
3    @Override
4    protected List<Availability> onPeriodicPresence(
5        Map<ParkingLotEnum, List<Boolean>> presenceByParkingLot) {
6      List<Availability> availabilityList =
7                          new ArrayList<Availability>();
8      for(Entry<ParkingLotEnum, List<Boolean>> parkingLot :
           presenceByParkingLot.entrySet()) {
9        int sum = 0;
10       for (Boolean presence : parkingLot.getValue())
11       { if (!presence) sum++; }
12       availabilityList.add(
13            new Availability(parkingLot.getKey(), sum);
14     } return availabilityList;
15   }}
```

Figure 5. ParkingAvailability context implementation.

```
1  public class ParkingEntrancePanelController extends
2                AbstractParkingEntrancePanelController {
3    @Override
4    protected void onParkingAvailability(Discover discover,
5          ParkingAvailabilityValue parkingAvailability) {
6      for(Availability availability :
7                        parkingAvailability.getValue()) {
8        String status = getStatus(availability);
9        discover.parkingEntrancePanels().whereLocation(
10             availability.getParkingLot()).update(status);
11     }}}
```

Figure 6. ParkingEntrancePanel controller implementation.

References

[1] Libelium. Smart City project in Santander to monitor Parking Free Slots. http://www.libelium.com/smart_santander_parking_smart_city.

[2] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. In ICSE '11, 2011.

[3] D. Cassou, J. Bruneau, C. Consel, and E. Balland. Towards a Tool-based Development Methodology for Pervasive Computing Applications. *IEEE TSE*, 38(6):1445–1463, 2012.

[4] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] J. Heidemann, F. Silva, and D. Estrin. Matching data dissemination algorithms to application requirements. In SenSys '03, 2003.

[7] X. Liu, Q. Huang, and Y. Zhang. Balancing push and pull for efficient information discovery in large-scale sensor networks. *IEEE Transactions on Mobile Computing*, 6(3):241–251, 2007.

[8] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[9] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman. A Taxonomy of Wireless Micro-Sensor Network Models. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(2):28–36, 2002.

[10] F. Zhu, M. W. Mutka, and L. M. Ni. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4(4):81–90, 2005.